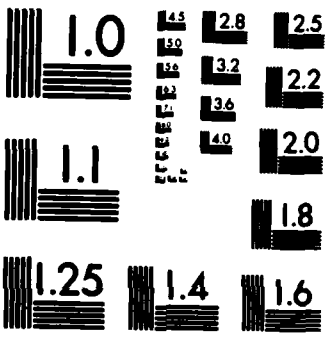
AD-A146 476 INTERNATIONAL LOGIC PROGRAMMING CONFERENCE (2ND) HELD AT UPPSALA SWEDEN ON 2-6 JULY 1984(U) OFFICE OF NAVAL RESEARCH LONDON (ENGLAND) J F BLACKBURN 27 JUL 84 1/1

UNCLASSIFIED ONRL-C-4-84 F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

AD-A146 476

ONR LONDON REPORT

C-4-84

OFFICE OF NAVAL RESEARCH

BRANCH OFFICE LONDON ENGLAND

SECOND INTERNATIONAL LOGIC PROGRAMMING CONFERENCE
J.F. BLACKBURN
27 JUL 1984

DTIC FILE COPY

UNITED STATES OF AMERICA

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

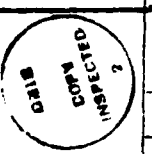
REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: C-4-84	2. GOVT ACCESSION NO.: AD-A146476	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Second International Logic Programming Conference	5. TYPE OF REPORT & PERIOD COVERED: Conference
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): J.F. Blackburn	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: US Office of Naval Research Branch Office London Box 39 FPO NY 09510	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE: 27 July 1984
	13. NUMBER OF PAGES: 8
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES



19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer programming
Programming languages
Logic programming
Sweden

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Second International Logic Programming Conference was held at Uppsala University, Sweden, from 2 through 6 July 1984. Presentations at the conference suggest that although logic programming is in the very early stages of exploitation, it is already becoming a useful tool.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SECOND INTERNATIONAL LOGIC PROGRAMMING CONFERENCE

The Second International Logic Programming Conference was devoted to discussions of logic programming, a relatively new concept in computer programming. The conference was held from 2 through 6 July at Uppsala University, Sweden; there were 228 participants from 22 countries. For copies of the proceedings, write to: Department of Computer Science, Uppsala University, Upmail, Box 2059, S-750 02 Uppsala, Sweden.

Background

The first computer programming language was the binary language of the machine, called machine code. The next step was to develop symbolic rather than numerical machine code, which was called Assembler language. After that came a series of higher level languages, including Fortran, Cobol, Basic, PL1, Apl, Algol, and more recently Pascal and Ada. In all of these the programmer must describe exactly how a result is to be computed. Such programs are largely made up of commands which specify actions to be performed, and are called imperative programming languages.

Logic programming languages are more descriptive than imperative programming languages, but they also have an imperative component. Such programs are primarily descriptive definitions of a set of relations or functions to be computed. The execution of such a program involves finding an output corresponding to a given input.

The first day of the conference was devoted mainly to a tutorial on logic programming, given by Keith Clark of Imperial College, London. This included a description of the logic programming language Prolog, the language Parlog (parallel programming in logic), and an introduction to expert systems.

The conference proper began on Tuesday morning. A welcoming address by S.-A. Tarnlund (Sweden) was followed by an invited address by R. Kowalski (UK), who spoke on problems of logic programming. He discussed three main problems but without offering specific solutions to them:

1. What is logic programming? He called it a compromise between theorem proving and programming.

2. What is the relation of logic programming to other forms of programming such as functional programming and object-oriented programming? And how should logic programming relate to concepts such as relational databases?

3. What problems need to be solved before logic programming "comes into its own"? As an example, he mentioned the problem of sorting. In a conventional imperative program a pointer can be made to point to the next item in a list when an item is removed, but at present in Prolog all the items up to the one removed are moved and stored. This results in a waste of time and memory space.

Application of Logic Programming I

Simpos

The first speaker in this category, Skigeyuki Takagi (Japan) described the overall design of Simpos (Sequential Inference Machine Programming and Operating System). The first major product of the Japanese Fifth Generation Computer Systems Project, the personal sequential inference machine (PSI or ¥) is under development. Takagi described the design of Simpos, the ¥'s programming and operating system; its major language, ESP (extended self-contained Prolog); and the development tools.

The major research theme of ¥ is to develop a logic-programming-based programming environment, including system programs. The basic design philosophy of Simpos is to build a super personal computer with database features and with Japanese natural-language processing under a logic-programming-based system design.

It is expected that by March 1985 such an operating/programming system with a good human interface will be in operation.

Prolog as a Tool for Optimizing Prolog Unities

Martin Nilsson (Sweden) discussed a unification procedure as a central part

of every Prolog implementation. He stated that a Prolog interpreter spends roughly half of its time unifying data structures. Consequently it is desirable to speed up unification as much as possible. He discussed generation of a speed-optimal unifier program and a comparison between the best and worst unifiers. A method for finding speed-optimal unifiers was given. The unifiers are generated by a Prolog program which is a declarative partial description of the unifier. The method has been applied to an experimental interpreter, for which some data were given.

Drawing Trees and Their Equations in Prolog

A detailed description of how to compute efficiently a drawing of Prolog trees with the smallest number of nodes was given by Jean Francois Pique (France). It is done using a system of equations described in "Prolog and Infinite Trees" (Colmerauer, 1982). Examples were given of finite and infinite trees in different domains.

Foundations of Logic Programs

A Logical Reconstruction of Prolog II

In a paper given jointly, M.H. Van Emden (Canada) and J.W. Lloyd, (Australia) took the view that a logic programming language is one in which a program is a first-order theory and computed answers are correct with respect to this theory. They then posed the question of whether Prolog II (Colmerauer, 1982) is a logic programming language and, if so, in what sense it is. Prolog II is regarded in Colmerauer's account as a system for rewriting possibly infinite trees. Unification is replaced by transformations on sets of equations.

Prolog II lacks occur check, but Colmerauer considers this lack an essential feature of the language, accounting for it in his tree-rewriting model. Van Emden and Lloyd answered the question of whether Prolog II is a logic programming language by making explicit Prolog II's theory of equality. Having done that they demonstrated that answers computed by Prolog II are correct with respect to a first-order theory consisting of the program plus the equality theory.

A brief account of Prolog II was given, followed by introduction of a general proof procedure which underlies both Prolog and Prolog II. Prolog was shown to be this general procedure plus the equality theory. Finally, Prolog II was shown to be essentially the general procedure plus a more complicated equality theory.

A Comparison of Two Logic Programming Languages

Szots Miklos (Hungary) compared Prolog and a new language called Lobo. After defining Lobo in detail, Miklos analyzed two examples dealing with planar covering problems--i.e., how a rectangle can be covered by given elements. Both languages were shown to be able to realize the same algorithms. The two languages are equivalent in the sense that both are suitable for defining all partial recursive functions. In that sense, both can be considered universal.

Lobo is nearer to traditional languages than is Prolog. Lobo does not use pattern matching, it can be compiled easily, and it is able to use traditional features of programming.

Computation Trees and Transformations of Logic Programs

Although they were not present, Olga Stepankova and Petr Stepanek (Czechoslovakia) sent in a paper introducing a new concept of computation trees of logic programs. The paper described three types of transformations that improve the structure of logic programs. Two natural measures of complexity are suggested by computation trees: the number of nodes called by recursion and the maximum number of and/or alterations on a branch. Stepankova and Stepanek showed that every logic program can be transformed to a program computing the same function, the computation tree of which has at most one called node and at most two alterations on every branch. Thus both measures of complexity collapse.

Applications of Logic Programming II

Semantic Interpretation for the Epistle System

Michael McCord (US) described Epistle, a natural-language processing system being developed at IBM Research. Its application is to text-critiquing, with criticism of grammar and style in documents. The Epistle grammar has broad coverage and is purely syntactic. McCord described a semantic interpretation component called Sem, written in Prolog, which will be useful in further developments for the system. Sem is based in part on previous work by McCord; it translates surface parses to logical forms in a single stage, in which there is interleaving of the processes of sense selection, slot filling, other types of modification, movement of nodes, and exercising of semantic constraints. Furthermore, the constraints used involve inference with world knowledge.

On Gapping Grammars

A discussion of gapping grammars was presented as a joint paper by Veronica Dahl and Harvey Abramson (Canada). A gapping grammar has rewriting rules of the form:

$$\alpha_1, \text{gap } (x_1), \alpha_2, \text{gap } (x_2), \ldots, \alpha_{n-1}, \text{gap } (x_{n-1}), \alpha_n \longrightarrow \beta$$

$$\alpha_1 \in V_N \cup V_T$$

$$G = [\text{gap } (x_1), \text{gap } (x_2), \ldots, \text{gap } (x_{n-1})]$$

$$x_1 \in V_T^*$$

$$\beta \in V_n^* \cup V_T^* \cup G^*$$

where V_T and V_N are the terminal and nonterminal vocabularies of the gapping grammar (GG). Intuitively, a GG rule allows one to deal with unspecified strings of terminal symbols called gaps, represented by X_1, $X_2, \ldots X_{n-1}$, in a given context of specified terminals and nonterminals, represented by α, $\alpha_2, \ldots$, α_n, and then to distribute them in the right-hand side β in any order. GGs are a generalization of Fernando Pereira's extraposition grammars where rules have the form:

$$\alpha_1, \text{gap } (x_1), \alpha_2, \text{gap } (x_2), \ldots, \text{gap } (x_{n-1}), \alpha_n \longrightarrow \beta, \text{gap } (x_1), \text{gap } (x_2), \ldots, \text{gap } (x_{n-1});$$

that is, the gaps are rewritten in their sequential order in the right-most positions of the rewriting rule. Examples were given in which extraposition grammars are not adequate, and alternative implementations of gapping grammars in logic were presented.

Logic Programming Languages

Eager and Lazy Enumerations in Concurrent Prolog

Hideki Hirakawa (Japan) discussed the possibility of using logic programming languages for AND-parallel and OR-parallel executions. Concurrent Prolog, designed by E. Shapiro, introduces an AND-parallelism and a limited OR-parallelism, i.e., a don't-care nondeterminism. The other aspect of OR-parallel execution--i.e., don't-know nondeterminism--is formalized as an "eager enumerate" operation on a set expression. Hirakawa described a computational model which provides the eager enumerate function to concurrent Prolog and showed its implementation in concurrent Prolog itself. He also showed that a lazy enumerate function can be implemented easily by introducing a bounded-buffer communication technique to the eager enumerator (Amamiya, 1984).

Incorporating Mutable Arrays Into Logic Programming

Logical terms are the only compound data structures in logic programming languages such as Prolog. Terms are sufficiently general so that no other data structures are needed. Restricted uses of terms correspond to the bits, character strings, arrays, and records of other programming languages. However, the computational overhead of using a very general data structure in specialized situations can be very high.

Side effects cannot be performed on logical terms, and the alternative of constructing new terms which differ slightly from the old can be very costly.

Lars-Henrik Eriksson (Sweden) suggested a way to alleviate the above shortcomings without losing logical clarity and purity. He and his colleague have introduced into LM-Prolog, a Prolog dialect running on Lisp machines, predicates for creating and manipulating arrays. These predicates could have been written as Horn clauses without the use of any primitives. They are implemented in terms of physical arrays and "virtual arrays" in a manner that is transparent to the user. For some uses of these predicates, it is possible for a compiler to produce code-performing array references and updates that are as good as those produced by compilers for traditional programming languages.

Equality, Types, Modules, and Generics for Logic

The original plan for logic programming called for the use of predicate logic as a programming language. Prolog only partially realizes this plan, since it has many features with no corresponding feature in first-order predicate logic. J.A. Goguen and J. Meseguer (US) suggested a system called Eqlog, which combines the technology of Prolog (its efficient implementation with unification and backtracking) with functional programming (in an efficient first-order rewrite-rule implementation) to yield facilities that exceed those of Prolog plus those of functional programming. Logical variables can be included in equations, giving the ability to find general solutions to equations over user-defined abstract data types. This new power is provided in a uniform and rigorous way by using "narrowing" from the theory of rewrite rules to get a complete implementation of equality. It can be seen as a special kind of resolution. Also, user-definable abstract data types and generic (i.e., parametrized) modules become available with a rigorous logical foundation. Eqlog also has a subsort facility that greatly increases its expressive power. Since Goguen and Meseguer's approach to generic modules and abstract data types relies on general results from the theories of specification languages and rewrite rules, it applies to ordinary unsorted Prolog and should also apply to other logic programming languages such as concurrent Prolog.

Logic Programming Methodology I

Unfold/Fold Transformation of Logic Programs

The unfold/fold transformation method is formulated for logic programs so that the transformation always preserves the equivalence of programs as defined by the least-model semantics. Hisao Tamaki (Japan) gave a detailed proof for the basic system. Some augmenting rules were also introduced, and the conditions for their safe application within the unfold/fold system were clarified. There are useful special cases of those rules in which application is always safe.

Bounded-Horizon, Success-Complete Restriction of Inference Programs

Michel Sintzoff (Belgium) dealt with the control of search in logic programming (Horn-clause inference) by the addition of restrictive predicates to rules so as to cut off all blind alleys without losing possible results. Criteria were proposed to ensure that additional premises allow results to be established without trial and error. These criteria require neither the introduction of special well-orderings nor the induction of limits of predicates. They take into account structural properties of bounded-length compositions of the original clauses and consequently are only sufficient.

An Efficient Bug-Location Algorithm

D.A. Plaisted (US) presented an efficient algorithm for locating bugs (errors) in Prolog. The algorithm, based on the method of Shapiro (1983) can be applied to any high-level programming language. The method is optimal

to within a constant factor for space, time, and number of queries to the user. This significantly improves the performance of Shapiro's method, which is not optimal for space or time and for which the number of queries depends on the branching factor of the computation. Since no current programming environment uses this method, it should be a significant aid to programmers in debugging software.

Architecture and Hardware for Logic Programming

OR-Parallelism on Applicative Architectures

A few years ago, Gary Lindstrom (US) introduced an abstract method for OR-parallel logic program execution--a method oriented toward applicative architectures. Central to this method is pipelined processing of streams of substitution data objects. At this meeting he addressed two implementation issues associated with the above approach: (1) the efficient representation of substitution data objects, and (2) a parallel unification algorithm compatible with this representation.

The approach to the first issue was use of a compact vectorized representation permitting indexed access of local variable bindings. Results on the second issue make use of a formulation of unification as a write-once database update problem, which can be efficiently implemented by a particular combination of applicative and imperative architectural features.

A Class of Architectures for a Prolog Machine

L.V. Kale (US) presented a view of the computation of Prolog programs that is suitable for expressing parallelism. He and his colleagues developed an idealized architecture which allows for exploiting most types of parallelisms. The architecture is based on an efficient broadcast link. The idealized architecture requires a lot of resources; therefore, various ways of mapping it onto practical topologies were sought. Types of parallelism that should be retained while making this approximation were discussed, and a class of architectures was developed that approximates the ideal. The parameters of this class were defined and criteria for evaluating them were given.

An Architecture for Parallel Logic Languages

An outline of an architecture to support the parallel execution of logic languages was presented by J.A. Crammond (UK). The implementation of a particular language, Parlog, was discussed. Special attention was given to its don't-care nondeterminism which allows both AND- and OR-parallelism and returns only one solution.

The main features described were the control structure and the binding environment. The proposed control structure uses processes that build an AND/OR tree tailored for guarded clauses. For the binding environment a unification algorithm was introduced which solves the problems of multiple occurrences of an instance of a variable in guard clauses.

A Highly Parallel Prolog Interpreter Based on the Generalized Data-Flow Model

Peter Kacsuk (Hungary) discussed a generalized data-flow model and its applications for constructing a highly parallel Prolog interpreter. The parallel Prolog interpreter is suitable for using advantages of OR- and AND-parallelism. Transformation of the AND/OR tree into a data-flow graph based on the generalized data-flow model was shown. Operator types needed for parallel evaluation of Prolog programs were explained in detail.

Application of Logic Programming III

A Prolog System for the Verification of Concurrent Processes Against Temporal Logic Specifications

E. Giovannetti (Italy) described a system, implemented in Prolog, for the verification of dynamic properties of concurrent processes. Description of

concurrent processes with asynchronous communication can be checked against dynamic behavior specifications expressed by temporal logic formulas, under the hypothesis that the whole concurrent system can be modeled by a nondeterministic finite automation.

Implementation for the basic components of the verifier was shown. Included were the model checkers for the chosen temporal logics, the symbolic simplifier, and the dynamic semantics of the description language.

Logical Levels of Problem Solving

A paper presented by Leon Sterling (Israel) demonstrated how clear, efficient problem-solving programs can be written in logic programming. The key point is the consideration of levels involved, both in the problem solving itself and in the underlying logic. Sterling identified three levels of knowledge necessary for intelligent problem solving: a level of domain knowledge, a level of methods and strategies, and a planning level. The approach suggested relates these levels to the distinction between object and meta languages. Two classes of programs were presented. First, single-level problem solvers were introduced; these are at the methods level and constitute a meta language of the problem domain. Second, flexible multi-level problem solvers were outlined which can be built as extensions of the single-level programs.

Logic Programming Methodology II

Using Symmetry for the Derivation of Logic Programs

In a programming calculus the formal development of a Horn-clause logic program implies a derivation of program clauses from a set of definitions, of data structures and computable functions, given in full predicate logic. A logic program is composed of a set of program clauses. Each program clause is derived separately from the definitions. The derivations differ mainly in structure for the different clauses. There are, however, cases when two program clauses in a program are similar (except for some small difference), so that they can be transformed into each other. Anna-Lena Johansson (Sweden) discussed ways to avoid constructing both the derivations, which can be quite lengthy. She further discussed how, after constructing a derivation of a program clause, one can answer the question of whether there is an analogous program clause. If there is, one needs to determine its appearance as well as the substitution on which to base an application of the substitution rule.

A Model Theory of Logic Programming Methodology

H. Sun (China) gave an axiom for a new version of the first-order language for mechanical theorem proving. It is called subgoal deduction language (SDL) and is used as a meta language for specification and derivation of logic programs as well as for representation of the knowledge necessary for program reasoning. A diversified relation system is defined as the semantic interpretation of SDL and logic programs. An example of an automatic derivation of a logic program from its specifications was given.

Foundations of Logic Programming II

A Unified Treatment of Resolution Strategies for Logic Programs

D.A. Wolfram (Australia) discussed a unified treatment of soundness and weak and strong completeness of various logic-program resolution strategies with respect to success and failure. The treatment is generalized and considerably simplified. This is made possible by using the full power of the unification theorem, which allows a reduction to a simple canonical case. The results can then be established in a straightforward manner. He also indicated how the unification theorem can be used to simplify the proof of the completeness of the negation as failure rule. Finally, he noted that the treatment applies to other clausal forms.

Applications of Logic Programming IV

Fame: A Prolog Program That Solves Problems in Combinatorics

Yoav Shoham (US) reported on Fame, a Prolog program that solves problems in combinatorics. He gave examples of the kind of problems that the program is able to solve:

1. Give a combinatorial argument for the following equalities: C(N-1, R) = (R+1)C(N, R+1) = (N-R)C(N, R).
2. Explain why the number of ways to put N indistinct objects into K distinct boxes is C(N+K+1, K-1).
3. How many ways are there to put N indistinct objects into K distinct boxes when every box receives at least one of the objects? How does this problem relate to the one before?
4. In an arrangement of 11 consecutive seats, how many ways are there to select four seats so that no two are adjacent? Explain your answer.
5. Give a combinatorial argument that Sigma (I from 0 to N, C(N, I) **2) = C(2N, N).

An advantage in using Prolog for solving such problems is its conciseness. One main reason for the conciseness of the program is the logic programming aspect of Prolog.

An Expert System for Computer Crash Analysis

Alan Littleford (US) described a large expert system that has been constructed using Prolog, which diagnoses computer-system crashes in a customer/field engineering environment.

He described the construction of the expert system and compared its implementation to some other similar schemes. Even though Prolog is a production-rule system, Littleford found it necessary to add an extra level of interpretation to meet some of the needs of the application.

Parlog for Discrete Event Simulation

A parallel-logic programming language called Parlog was described by Steve Gregory (UK). The Parlog language features both OR- and AND-parallelism. It was designed to simulate a system by a network of parallel processes communicating by messages. Real time is replaced by a central simulated clock. The Parlog language is based on Horn clauses, and it differs from Prolog in two important respects: don't-care nondeterminism and the use of modes. These features make possible the concurrent evaluation of conjoined relation calls--i.e., AND-parallelism, with stream communication between the calls. Each relation call is evaluated as a process; shared variables act as one-way communication channels along which messages are sent by incremental binding to lists.

Conclusion

The number of people attending this Second International Logic Programming Conference is a clear indication of the influence the meeting has achieved. Logic programming is in the very early stages of exploitation, but it is already becoming a useful tool at this early stage. Its use in programming expert systems has shown that simple expert systems can solve practical problems. Several good examples given during the conference were the system for computer crash analysis described by Littleford; the Fame system for solving problems in combinatorics, described by Shoham; and Parlog for parallel logic programming, described by Gregory.

A highly significant development is the combining of logic programming and functional programming, as described by Goguen and Meseguer. The combining of the two systems in Eqlog yields capabilities that exceed those of Prolog and functional programming combined; one gets the logic capability of Prolog and the ability to handle numerical problems through functional programming.

Logic programming has great potential and has been given greatly added impetus by the Japanese decision to make their Fifth Generation computers logic machines.

References

Amamiya, M., and R. Hasegawa, "Dataflow Computing and Eager and Lazy Evaluations," *New Generation Computing*, 2 (1984), 105-129.

Colmerauer, A., "Prolog and Infinite Trees," *Logic Programming*, APIC Studies in Data Processing No. 16 (London: Academic Press, 1982), 231-251.

END

FILMED

11-84

DTIC